# Part 2

# Protocols: The recipes of cryptography

You are now entering the second part of this book, which is going to make use of most of what you've learned in the first part. Think about it this way: if the cryptographic primitives you've learned about were the basic ingredients of cryptography, you're now about to learn some recipes. And there's a lot to cook! While Caesar might have only been interested in encrypting his communications, today cryptography is all over the place, and it's quite hard to keep track of it all.

In chapter 9, 10, and 11, I show you where you are most likely to run into cryptography and how cryptography is used to solve real-world problems; that is, how cryptography encrypts communications and how it authenticates participants in protocols. For the most part, that's what cryptography is about. Participants will be numerous or few, and made of bits or flesh. As you'll quickly realize, real-world cryptography is about tradeoffs and, based on the context, solutions will differ.

Chapter 12 and 13 take you into two quickly evolving fields of cryptography: cryptocurrencies and hardware cryptography. The former topic has been ignored by most books on cryptography. (I believe that this book, *Real-World Cryptography*, is the first cryptography book to include a chapter on cryptocurrencies.) The latter topic, hardware cryptography, is often overlooked too; cryptographers often assume that their primitives and protocols run in a trusted environment, which is less and less the case. Hardware cryptography is about pushing the boundaries of where cryptography can run and providing security assurances when attackers are getting closer and closer to you.

In chapters 14 and 15, I touch on the bleeding edge: what's not here yet but will be and what's sort of here. You'll learn about postquantum cryptography, which is a field of cryptography that might be useful, depending on if we, as a human species, invent scalable quantum computers. These quantum computers, based on novel paradigms coming from the realm of quantum physics, could revolutionize research and, perhaps, even break our crypto . . . You'll also learn about what I call "next-generation cryptography," cryptographic primitives that have rarely seen the light of day but that you will most likely see more frequently as these get studied, become more efficient, and get adopted by application designers. Finally, I conclude the book in chapter 16 with some final remarks on real-world cryptography and some words on ethics.

9

# Secure transport

**This chapter covers**

- Secure transport protocols
- The Transport Layer Security (TLS) protocol
- The Noise protocol framework

The heaviest use of cryptography today is most probably to encrypt communications. After all, cryptography was invented for this purpose. To do this, applications generally do not make use of cryptographic primitives like authenticated encryption directly, but instead use much more involved protocols that abstract the use of the cryptographic primitives. I call these protocols *secure transport protocols*, for lack of a better term.

In this chapter, you will learn about the most widely used secure transport protocol: the Transport Layer Security (TLS) protocol. I will also lightly cover other secure transport protocols and how they differ from TLS.

## 9.1 The SSL and TLS secure transport protocols

In order to understand why *transport protocols* (protocols used to encrypt communications between machines) are a thing, let's walk through a motivating scenario. When you enter, say, `http://example.com` in your web browser, your browser uses a number of protocols to connect to a web server and to retrieve the page you requested. One of those is the *Hypertext Transfer Protocol* (HTTP), which your browser uses to tell the web server on the other side which page it is interested in. HTTP uses a human-readable format. This means that you can look at the HTTP messages that are being sent and received over the wire and read them without the help of any other tool. But this is not enough for your browser to communicate to the web server.

HTTP messages are encapsulated into other types of messages, called *TCP frames*, which are defined in the Transmission Control Protocol (TCP). TCP is a binary protocol, and thus, it is not human-readable: you need a tool to understand the fields of a TCP frame. TCP messages are further encapsulated using the Internet Protocol (IP), and IP messages are further encapsulated using something else. This is known as the *Internet protocol suite*, and as it is the subject of many books, I won't go much further into this.

Back to our scenario, as there's a confidentiality issue that we need to talk about. Anyone sitting on the wire in between your browser and the web server of example.com has an interesting position: they can passively observe and read your requests as well as the server's responses. Worse, MITM attackers can also actively tamper and reorder messages. This is not great.

Imagine your credit card information leaking out every time you buy something on the internet, your passwords being stolen when you log into a website, your pictures and private messages pilfered as you send those to your friends, and so forth. This scared enough people that in the 1990s, the predecessor of TLS—the *Secure Sockets Layer* (SSL) *protocol*—was born. While SSL can be used in different kinds of situations, it was first built by and for web browsers. As such, it started being used in combination with HTTP, extending it into the *Hypertext Transfer Protocol Secure* (HTTPS). HTTPS now allowed browsers to secure their communications to the different websites they visited.

### 9.1.1 From SSL to TLS

Although SSL was not the only protocol that attempted to secure some of the web, it did attract most of the attention and, with time, has become the de facto standard. But this is not the whole story. Between the first version of SSL and what we currently use today, a lot has happened. All versions of SSL (the last being SSL v3.0) were broken due to a combination of bad design and bad cryptographic algorithms. (Many of the attacks have been summarized in RFC 7457.)

After SSL 3.0, the protocol was officially transferred to the Internet Engineering Task Force (IETF), the organization in charge of publishing *Request For Comments* (RFCs) standards. The name SSL was dropped in favor of TLS, and TLS 1.0 was released in 1999 as RFC 2246. The most recent version of TLS is TLS 1.3, specified in RFC 8446 and published in 2018. TLS 1.3, unlike its predecessor, stems from a solid collaboration between the industry and academia. Yet, today, the internet is still divided between many different versions of SSL and TLS as servers have been slow to update.

**NOTE** There's a lot of confusion around the two names SSL and TLS. The protocol is now called *TLS*, but many articles and even libraries still choose to use the term *SSL*.

TLS has become more than just the protocol securing the web; it is now used in many different scenarios and among many different types of applications and devices as a protocol to secure communications. Thus, what you will learn about TLS in this chapter is not only useful for the web, but also for any scenario where communications between two applications need to be secure.

### 9.1.2 Using TLS in practice

How do people use TLS? First let's define some terms. In TLS, the two participants that want to secure their communications are called a *client* and a *server*. It works the same way as with other network protocols like TCP or IP: the client is the one that initiates the connection, and the server is the one that waits for one to be initiated. A TLS client is typically built from

- *Some configuration*—A client is configured with the versions of SSL and TLS that it wants to support, cryptographic algorithms that it is willing to use to secure the connection, ways it can authenticate servers, and so on.
- *Some information about the server it wants to connect to*—It includes at least an IP address and a port, but for the web, it often includes a fully qualified domain name instead (like example.com).

Given these two arguments, a client can initiate a connection with a server to produce a secure *session*, a channel that both the client and the server can use to share encrypted messages with each other. In some cases, a secure session cannot successfully be created and fails midway. For example, if an attacker attempts to tamper with the connection or if the server's configuration is not compatible with the client's (more on that later), the client fails to establish a secure session.

A TLS server is often much simpler as it only takes a configuration, which is similar to the client's configuration. A server then waits for clients to connect to it in order to produce a secure session. In practice, using TLS on the client side can be as easy as the following listing demonstrates (that is, if you use a programming language like Golang).

**Listing 9.1 A TLS client in Golang**

```
import "crypto/tls"

func main() {
    destination := "google.com:443"
     TLSconfig := &tls.Config{}
     conn, err := tls.Dial("tcp", destination, TLSconfig)
    if err != nil {
        panic("failed to connect: " + err.Error())
```

**The fully qualified domain name and the server's port (443 is the default port for HTTPS).**

**An empty config serves as the default configuration.**

```
    }
    conn.Close()
}
```

How does the client know that the connection it established is really with google.com and not some impersonator? By default, Golang's TLS implementation uses your operating system's configuration to figure out how to authenticate TLS servers. (Later in this chapter, you will learn exactly how the authentication in TLS works.) Using TLS on the server side is pretty easy as well. The following listing shows how simple this is.

**Listing 9.2 A TLS server in Golang**

```
import (
    "crypto/tls"
    "net/http"
)

func hello(rw http.ResponseWriter, req *http.Request) {
    rw.Write([]byte("Hello, world\n"))
}

func main() {
    config := &tls.Config{
         MinVersion: tls.VersionTLS13,
     }

    http.HandleFunc("/", hello)

    server := &http.Server{
         Addr:        ":8080",
         TLSConfig: config,
     }

    cert := "cert.pem"
    key  := "key.pem"
    err := server.ListenAndServeTLS(cert, key)
     if err != nil {
        panic(err)
    }
}
```

**A solid minimal configuration for a TLS 1.3 server**

**Serves a simple page displaying "Hello, world".**

**An HTTPS server starts on port 8080.**

**Some .pem files containing a certificate and a secret key (more on this later)**

Golang and its standard library do a lot for us here. Unfortunately, not all languages' standard libraries provide easy-to-use TLS implementations, if they provide a TLS implementation at all, and not all TLS libraries provide secure-by-default implementations! For this reason, configuring a TLS server is not always straightforward, depending on the library. In the next section, you will learn about the inner workings of TLS and its different subtleties.

**NOTE** TLS is a protocol that works on top of TCP. To secure UDP connections, we can use DTLS (*D* is for *datagram*, the term for UDP messages), which is fairly similar to TLS. For this reason, I ignore DTLS in this chapter.

## 9.2 How does the TLS protocol work?

As I said earlier, today TLS is the de facto standard to secure communications between applications. In this section, you will learn more about how TLS works underneath the surface and how it is used in practice. You will find this section useful for learning how to use TLS properly and also for understanding how most (if not all) secure transport protocols work. You will also find out why it is hard (and strongly discouraged) to redesign or reimplement such protocols.

At a high level, TLS is split into two phases as noted in the following list. Figure 9.1 illustrates this idea.

- *A handshake phase*—A secure communication is negotiated and created between two participants.
- *A post-handshake phase*—Communications are encrypted between the two participants.

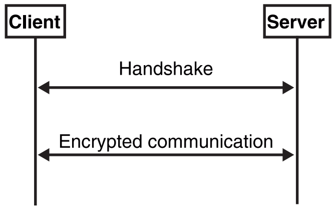


**Figure 9.1 At a high level, secure transport protocols first create a secure connection during a handshake phase. After that, applications on both sides of the secure connection can communicate securely.**

At this point, because you learned about hybrid encryption in chapter 6, you should have the following (correct) intuition about how these two steps works:

- *The handshake is, at its core, simply a key exchange.* The handshake ends up with the two participants agreeing on a set of symmetric keys.
- *The post-handshake phase is purely about encrypting messages between participants.* This phase uses an authenticated encryption algorithm and the set of keys produced at the end of the handshake.

Most transport security protocols work this way, and the interesting parts of these protocols always lie in the handshake phase. Next, let's take a look at the handshake phase.

### 9.2.1 The TLS handshake

As you've seen, TLS is (and most transport security protocols are) divided into two parts: a *handshake* and a *post-handshake* phase. In this section, you'll learn about the handshake first. The handshake itself has four aspects that I want to tell you about:

- *Negotiation*—TLS is highly configurable. Both a client and a server can be configured to negotiate a range of SSL and TLS versions as well as a menu of acceptable cryptographic algorithms. The negotiation phase of the handshake aims at finding common ground between the client's and the server's configurations in order to securely connect the two peers.

- *Key exchange*—The whole point of the handshake is to perform a key exchange between two participants. Which key exchange algorithm to use? This is one of the things decided as part of the client/server negotiation process.
- *Authentication*—As you learned in chapter 5 on key exchanges, it is trivial for MITM attackers to impersonate any side of a key exchange. Due to this, key exchanges must be authenticated. Your browser must have a way to ensure that it is talking to google.com, for example, and not your internet service provider (ISP).
- *Session resumption*—As browsers often connect to the same websites again and again, key exchanges can be costly and can slow down a user's experience. For this reason, mechanisms to fast-track secure sessions without redoing a key exchange are integrated into TLS.

This is a comprehensive list! As fast as greased lightning, let's start with the first item.

#### Negotiation in TLS: What version and what algorithms?

Most of the complexity in TLS comes from the negotiation of the different moving parts of the protocol. Infamously, this negotiation has also been the source of many issues in the history of TLS. Attacks like FREAK, LOGJAM, DROWN, and others took advantage of weaknesses present in older versions to break more recent versions of the protocol (sometimes even when the server did not support older versions!). While not all protocols have versioning or allow for different algorithms to be negotiated, SSL/TLS was designed for the web. As such, SSL/TLS needed a way to maintain backward compatibility with older clients and servers that could be slow to update.

This is what happens on the web today: your browser might be recent and up-to-date and made to support TLS version 1.3, but when visiting some old web page, chances are that the server behind it only supports TLS versions up to 1.2 or 1.1 (or worse). Vice-versa, many websites must support older browsers, which translates into supporting older versions of TLS (as some users are still stuck in the past).

#### Are older versions of SSL and TLS secure?

Most versions of SSL and TLS have security issues, except for TLS versions 1.2 and 1.3. Why not just support the latest version (1.3) and call it a day? The reason is that some companies support older clients that can't easily be updated. Due to these requirements, it is not uncommon to find libraries implementing mitigations to known attacks in order to securely support older versions. Unfortunately, these mitigations are often too complex to implement correctly.

For example, well-known attacks like Lucky13 and Bleichenbacher98 have been rediscovered again and again by security researchers in various TLS implementations that had previously attempted to fix the issues. Although it is possible to mitigate a number of attacks on older TLS versions, I would recommend against it, and I am not the only one telling you this. In March 2021, the IETF published RFC 8996: "Deprecating TLS 1.0 and TLS 1.1," effectively making the deprecation official.

Negotiation starts with the client sending a first request (called a *ClientHello*) to the server. The ClientHello contains a range of supported SSL and TLS versions, a suite of cryptographic algorithms that the client is willing to use, and some more information that could be relevant for the rest of the handshake or for the application. The suite of cryptographic algorithms include

- *One or more key exchange algorithms*—TLS 1.3 defines the following algorithms allowed for negotiations: ECDH with P-256, P-384, P-521, X25519, X448, and FFDH with the groups defined in RFC 7919. I talked about all of these in chapter 5. Previous versions of TLS also offered RSA key exchanges (covered in chapter 6), but they were removed in the last version.
- *Two (for different parts of the handshake) or more digital signature algorithms*—TLS 1.3 specifies RSA PKCS#1 version 1.5 and the newer RSA-PSS, as well as more recent elliptic curve algorithms like ECDSA and EdDSA. I talked about these in chapter 7. Note that digital signatures are specified with a hash function, which allows you to negotiate, for example, RSA-PSS with either SHA-256 or SHA-512.
- *One or more hash functions to be used with HMAC and HKDF*—TLS 1.3 specifies SHA-256 and SHA-384, two instances of the SHA-2 hash function. (You learned about SHA-2 in chapter 2.) This choice of hash function is unrelated to the one used by the digital signature algorithm. As a reminder, HMAC is the message authentication code you learned in chapter 3, and HKDF is the key derivation function we covered in chapter 8.
- *One or more authenticated encryption algorithms*—These can include AES-GCM with keys of 128 or 256 bits, ChaCha20-Poly1305, and AES-CCM. I talked about all of these in chapter 4.

The server then responds with a *ServerHello* message, which contains one of each type of cryptographic algorithm, cherry-picked from the client's selection. The following illustration depicts this response.

HELLO! I WANT TO CONNECT WITH TLS 1.3. I SUPPORT X448 AND X25519 FOR KEY EXCHANGES. AES-GCM AND CHACHA20-POLY1305 FOR AUTHENTICATED ENCRYPTION. ETC.
HELLO! FOR SURE. LET'S DO X25519 FOR KEY EXCHANGE, AES-GCM FOR AUTHENTICATED ENCRYPTION, ETC.

If the server is unable to find an algorithm it supports, it aborts the connection. Although in some cases, the server does not have to abort the connection and can ask the client to provide more information instead. To do this, the server replies with a message called a *HelloRetryRequest*, asking for the missing piece of information. The client can then resend its ClientHello, this time with the added requested information.

### TLS AND FORWARD-SECURE KEY EXCHANGES

The key exchange is the most important part of the TLS handshake! Without it, there's obviously no symmetric key being negotiated. But for a key exchange to happen, the client and the server must first trade their respective public keys.

In TLS 1.2 and previous versions, the client and the server start a key exchange only after both participants agree on which key exchange algorithm to use. This happens during a negotiation phase. TLS 1.3 optimizes this flow by attempting to do both the negotiation and the key exchange at the same time: the client speculatively chooses a key exchange algorithm and sends a public key in the first message (the ClientHello). If the client fails to predict the server's choice of key exchange algorithm, then the client falls back to the outcome of the negotiation and sends a new ClientHello containing the correct public key. The following steps describe how this might look. I illustrate the difference in figure 9.2.

1. The client sends a TLS 1.3 ClientHello message announcing that it can do either an X25519 or an X448 key exchange. It also sends an X25519 public key.
2. The server does not support X25519 but does support X448. It sends a HelloRetryRequest to the client announcing that it only supports X448.
3. The client sends the same ClientHello but with an X448 public key instead.
4. The handshake goes on.

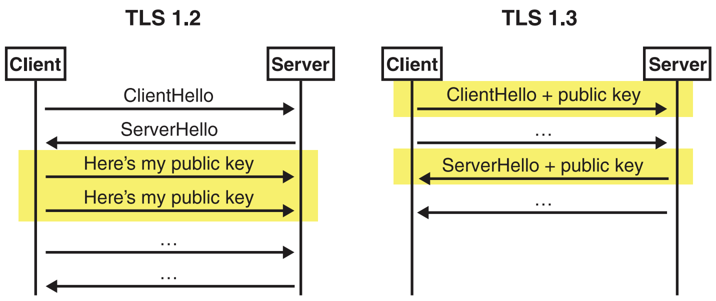


**Figure 9.2 In TLS 1.2, the client waits for the server to choose which key exchange algorithm to use before sending a public key. In TLS 1.3, the client speculates on which key exchange algorithm(s) the server will settle on and preemptively sends a public key (or several) in the first message, potentially avoiding an extra round trip.**

TLS 1.3 is full of such optimizations, which are important for the web. Indeed, many people worldwide have unstable or slow connections, and it is important to keep non-application communication to the bare minimum required. Furthermore, in TLS 1.3

(and unlike previous versions of TLS), all key exchanges are *ephemeral*. This means that for each new session, the client and the server both generate new key pairs, then get rid of them as soon as the key exchange is done. This provides *forward secrecy* to the key exchange: a compromise of the long-term keys of the client or the server, which won't allow an attacker to decrypt this session as long as the ephemeral private keys were safely deleted.

Imagine what would happen if, instead, a TLS server used a single private key for every key exchange it performs with its clients. By performing ephemeral key exchanges and getting rid of private keys as soon as a handshake ends, the server protects against such attackers. I illustrate this in figure 9.3.

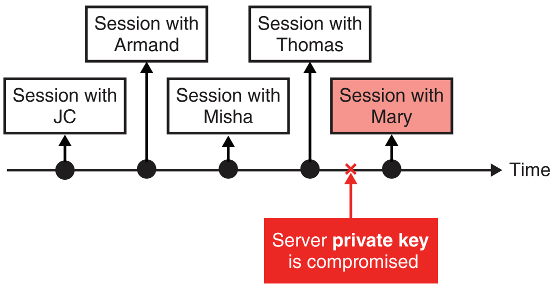


**Figure 9.3 In TLS 1.3, each session starts with an ephemeral key exchange. If a server is compromised at some point in time, no previous sessions will be impacted.**

**Exercise**

A compromise of the server's private key at some point in time would be devastating as MITM attackers would then be able to decrypt all previously recorded conversations. Do you understand how this can happen?

Once ephemeral public keys are traded, a key exchange is performed, and keys can be derived. TLS 1.3 derives different keys at different points in time to encrypt different phases with independent keys.

The first two messages, the ClientHello and the ServerHello, cannot be encrypted because no public keys were traded at this point. But after that, as soon as the key exchange happens, TLS 1.3 encrypts the rest of the handshake. (This is unlike previous versions of TLS that did not encrypt any of the handshake messages.)

To derive the different keys, TLS 1.3 uses HKDF with the hash function negotiated. HKDF-Extract is used on the output of the key exchange to remove any biases, while HKDF-Expand is used with different `info` parameters to derive the encryption keys. For example, `tls13 c hs traffic` (for "client handshake traffic") is used to derive symmetric keys for the client to encrypt to the server during the handshake, and `tls13 s ap traffic` (for "server application traffic") is used to derive symmetric keys for the server to encrypt to the client after the handshake. Remember though, *unauthenticated* key exchanges are insecure! Next, you'll see how TLS addresses this.

### TLS authentication and the web public key infrastructure

After some negotiations and after the key exchange has taken place, the handshake must go on. What happens next is the other most important part of TLS—*authentication*. You saw in chapter 5 on key exchanges that it is trivial to intercept a key exchange and impersonate one or both sides of the key exchange. In this section, I'll explain how your browser cryptographically validates that it is talking to the right website and not to an impersonator. But, first, let's take a step back. There is something I haven't told you yet. A TLS 1.3 handshake is actually split into three different stages (as figure 9.4 illustrates):

1. *Key exchange*—This phase contains the *ClientHello* and *ServerHello* messages that provide some negotiation and perform the key exchange. All messages including handshake messages after this phase are encrypted.
2. *Server parameters*—Messages in this phase contain additional negotiation data from the server. This is negotiation data that does not have to be contained in the first message of the server and that could benefit from being encrypted.
3. *Authentication*—This phase includes authentication information from both the server and the client.

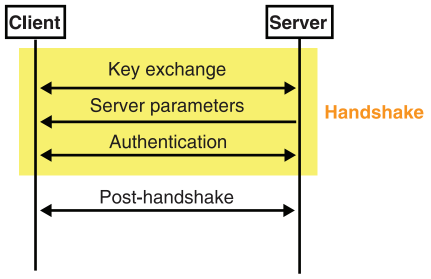


**Figure 9.4** A TLS 1.3 handshake is divided into three phases: the key exchange phase, the server parameters phase, and (finally) the authentication phase.

On the web, authentication in TLS is usually one-sided. Only the browser verifies that google.com, for example, is indeed google.com, but google.com does not verify who you are (or at least not as part of TLS).

#### Mutually-authenticated TLS

Client authentication is often delegated to the application layer for the web, most often via a form asking you for your credentials. That being said, client authentication can also happen in TLS if requested by the server during the server parameters phase. When both sides of the connection are authenticated, we talk about *mutually-authenticated TLS* (sometimes abbreviated as mTLS).

Client authentication is done the same way as server authentication. This can happen at any point after the authentication of the server (for example, during the handshake or in the post-handshake phase).

Let's now answer the question, "When connecting to google.com, how does your browser verify that you are indeed handshaking with google.com?" The answer is by using the *web public key infrastructure (web PKI)*.

You learned about the concept of public key infrastructure in chapter 7 on digital signatures, but let me briefly reintroduce this concept as it is quite important in understanding how the web works. There are two sides to the web PKI. First, browsers must trust a set of root public keys that we call *certificate authorities* (CAs). Usually, browsers will either use a hardcoded set of trusted public keys or will rely on the operating system to provide them.

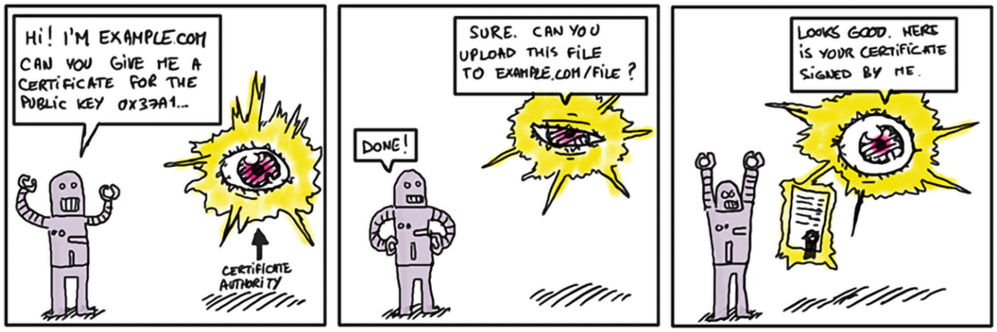
**The web PKI**

For the web, there exist hundreds of these CAs that are independently run by different companies and organizations across the world. It is quite a complex system to analyze, and these CAs can sometimes also sign the public keys of intermediate CAs that, in turn, also have the authority to sign the public keys of websites. For this reason, organizations like the *Certification Authority Browser Forum* (CA/Browser Forum) enforce rules and decide when new organizations can join the set of trusted public keys or when a CA can no longer be trusted and must be removed from that set.

Second, websites that want to use HTTPS must have a way to obtain a certification (a signature of their signing public key) from these CAs. In order to do this, a website owner (or a webmaster, as we used to say) must prove to a CA that they own a specific domain.

**NOTE** Obtaining a certificate for your own website used to involve a fee. This is no longer the case as CAs like Let's Encrypt provide certificates for free.

To prove that you own example.com, for example, a CA might ask you to host a file at example.com/some_path/file.txt that contains some random numbers generated for your request. The following comic strip shows this exchange.


HI! I'M EXAMPLE.COM CAN YOU GIVE ME A CERTIFICATE FOR THE PUBLIC KEY 0X37A1...
CERTIFICATE AUTHORITY
SURE. CAN YOU UPLOAD THIS FILE TO EXAMPLE.COM/FILE ?
DONE!
LOOKS GOOD. HERE IS YOUR CERTIFICATE SIGNED BY ME.

After this, a CA can provide a signature over the website's public key. As the CA's signature is usually valid for a period of years, we say that it is over a long-term signing public key (as opposed to an ephemeral public key). More specifically, CAs do not actually sign public keys, but instead they sign *certificates* (more on this later). A certificate contains the long-term public key, along with some additional important metadata like the web page's domain name.

To prove to your browser that the server it is talking to is indeed google.com, the server sends a *certificate chain* as part of the TLS handshake. The chain comprises

- Its own leaf certificate, containing (among others) the domain name (google.com, for example), Google's long-term signing public key, as well as a CA's signature
- A chain of intermediate CA certificates from the one that signed Google's certificate to the root CA that signed the last intermediate CA

This is a bit wordy so I illustrated this in figure 9.5.

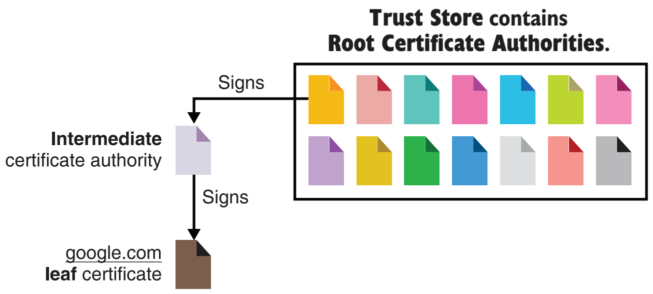


**Figure 9.5 Web browsers only have to trust a relatively small set of root CAs in order to trust the whole web. These CAs are stored in what is called a *trust store*. In order for a website to be trusted by a browser, the website must have its leaf certificate signed by one of these CAs. Sometimes root CAs only sign intermediate CAs, which, in turn, sign other intermediate CAs or leaf certificates. This is what's known as the web PKI.**

The certificate chain is sent in a certificate TLS message by the server and by the client as if the client has been asked to authenticate. Following this, the server can use its certified long-term key pair to sign all handshake messages that have been received and previously sent in what is called a *CertificateVerify* message. Figure 9.6 reviews this flow, where only the server authenticates itself.

The signature in the CertificateVerify message proves to the client what the server has so far seen. Without this signature, a MITM attacker could intercept the server's handshake messages and replace the ephemeral public key of the server contained in the ServerHello message, allowing the attacker to successfully impersonate the server. Take a few moments to understand why an attacker cannot replace the server's ephemeral public key in the presence of the CertificateVerify signature.

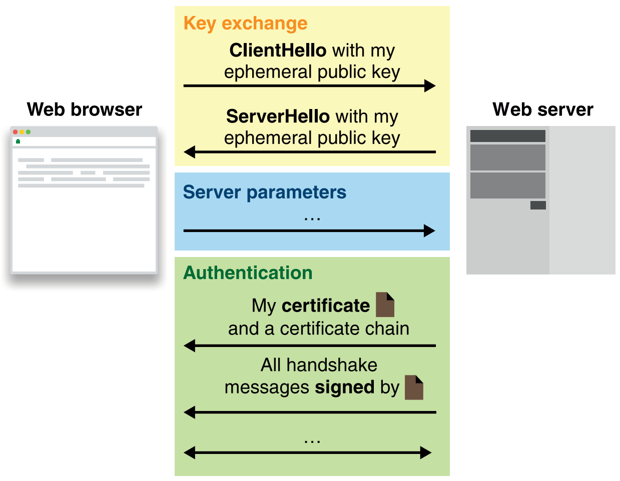

Web browser
Key exchange
ClientHello with my ephemeral public key
ServerHello with my ephemeral public key
Server parameters
...
Authentication
My certificate and a certificate chain
All handshake messages signed by
...
Web server

**Figure 9.6** **The authentication part of a handshake starts with the server sending a certificate chain to the client. The certificate chain starts with the leaf certificate (the certificate containing the website's public key and additional metadata like the domain name) and ends with a root certificate that is trusted by the browser. Each certificate contains a signature from the certificate above it in the chain.**

## Story time

A few years ago, I was hired to review a custom TLS protocol made by a large company. It turned out that their protocol had the server provide a signature that did not cover the ephemeral key. When I told them about the issue, the whole room went silent for a full minute. It was, of course, a substantial mistake: an attacker who could have intercepted the custom handshake and replaced the ephemeral key with its own would have successfully impersonated the server.

The lesson here is that it is important not to reinvent the wheel. Secure transport protocols are hard to get right, and if history has shown anything, they can fail in many unexpected ways. Instead, you should rely on mature protocols like TLS and make sure you use a popular implementation that has received a substantial amount of public attention.

Finally, in order to officially end the handshake, both sides of the connection must send a *Finished* message as part of the authentication phase. A Finished message contains an authentication tag produced by HMAC, instantiated with the negotiated hash function for the session. This allows both the client and the server to tell the other side, "These are all the messages I have sent and received in order during this handshake." If the handshake is intercepted and tampered with by MITM attackers, this integrity check allows the participants to detect and abort the connection. This is especially useful as some handshakes modes are *not* signed (more on this later).

Before heading to a different aspect of the handshake, let's look at X.509 certificates. They are an important detail of many cryptographic protocols.

### AUTHENTICATION VIA X.509 CERTIFICATES

While certificates are optional in TLS 1.3 (you can always use plain keys), many applications and protocols, not just the web, make heavy use of them in order to certify additional metadata. Specifically, the X.509 certificate standard version 3 is used.

X.509 is a pretty old standard that was meant to be flexible enough to be used in a multitude of scenarios: from email to web pages. The X.509 standard uses a description language called *Abstract Syntax Notation One* (ASN.1) to specify information contained in a certificate. A data structure described in ASN.1 looks like this:

```
Certificate  ::=  SEQUENCE  {
    tbsCertificate       TBSCertificate,
    signatureAlgorithm   AlgorithmIdentifier,
    signatureValue       BIT STRING  }
```

You can literally read this as a structure that contains three fields:

- `tbsCertificate`—The to-be-signed certificate. This contains all the information that one wants to certify. For the web, this can contain a domain name (google.com, for example), a public key, an expiration date, and so on.
- `signatureAlgorithm`—The algorithm used to sign the certificate.
- `signatureValue`—The signature from a CA.

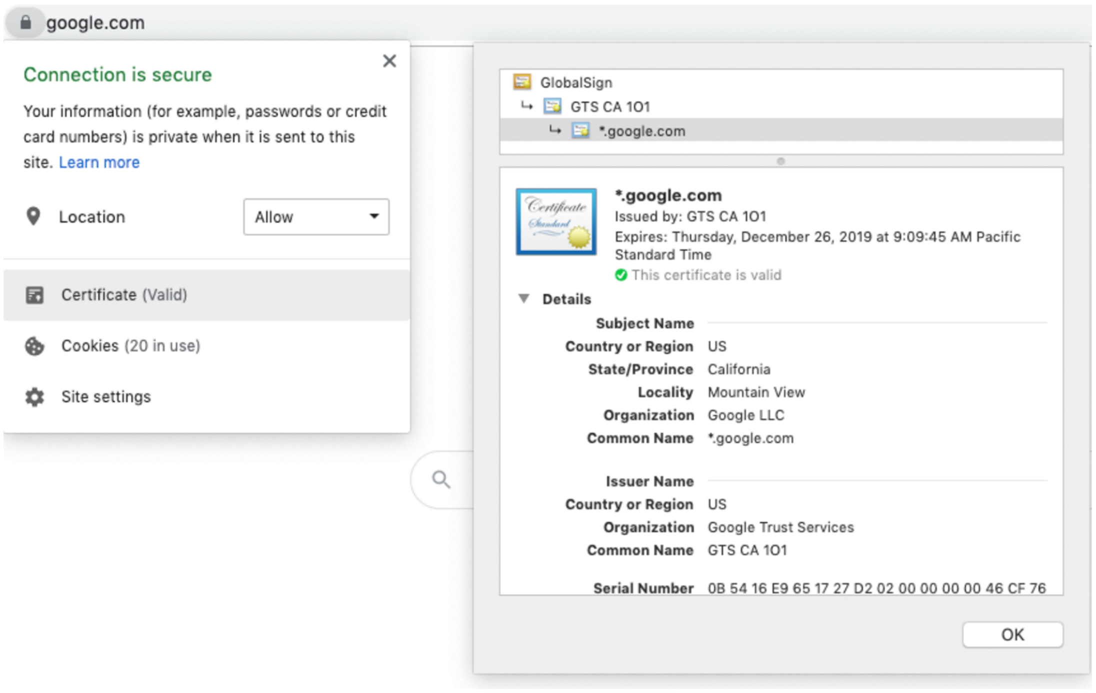
**Exercise**

The values `signatureAlgorithm` and `signatureValue` are not contained in the actual certificate, `tbsCertificate`. Do you know why?

You can easily check what's in an X.509 certificate by connecting to any website using HTTPS and then using your browser functionalities to observe the certificate chain sent by the server. See figure 9.7 for an example.

You might encounter X.509 certificates as .pem files, which is some base64-encoded content surrounded by some human-readable hint of what the base64-encoded data contains (here, a certificate). The following snippet represents the content of a certificate in a .pem format:

```
-----BEGIN CERTIFICATE-----
MIIJQzCCCCugAwIBAgIQC1QW6WUXJ9ICAAAAAEbPdjANBgkqhkiG9w0BAQsFADBC
MQswCQYDVQQGEwJVUzEeMBwGA1UEChMVR29vZ2xlIFRydXN0IFNlcnZpY2VzMRMw
EQYDVQQDEwpHVFMgQ0EgMU8xMB4XDTE5MTAwMzE3MDk0NVoXDTE5MTIyNjE3MDk0
NVowZjELMAkGA1UEBhMCVVMxEzARBgNVBAgTCkNhbGlmb3JuaWExFjAUBgNVBAcT
[...]
vaoUqelfNJJvQjJbMQbSQEp9y8EIi4BnWGZjU6Q+q/3VZ7ybR3cOzhnaLGmqiwFv
4PNBdnVVfVbQ9CxRiplKVzZSnUvypgBLryYnl6kquh1AJS5gnJhzogrz98IiXCQZ
c7mkvTKgCNIR9fedIus+LPHCSD7zUQTgRoOmcB+kwY7jrFqKn6thTjwPnfB5aVNK
dl0nq4fcF8PN+ppgNFbwC2JxX08L1wEFk2LvDOQgKqHR1TRJ0U3A2gkuMtf6Q6au
3KBzGW6l/vt3coyyDkQKDmT61tjwy5k=
-----END CERTIFICATE-----
```


google.com

Connection is secure

Your information (for example, passwords or credit card numbers) is private when it is sent to this site. Learn more

Location: Allow

Certificate (Valid)

Cookies (20 in use)

Site settings

GlobalSign
↳ GTS CA 1O1
↳ *.google.com

*.google.com
Issued by: GTS CA 1O1
Expires: Thursday, December 26, 2019 at 9:09:45 AM Pacific Standard Time
This certificate is valid

Details

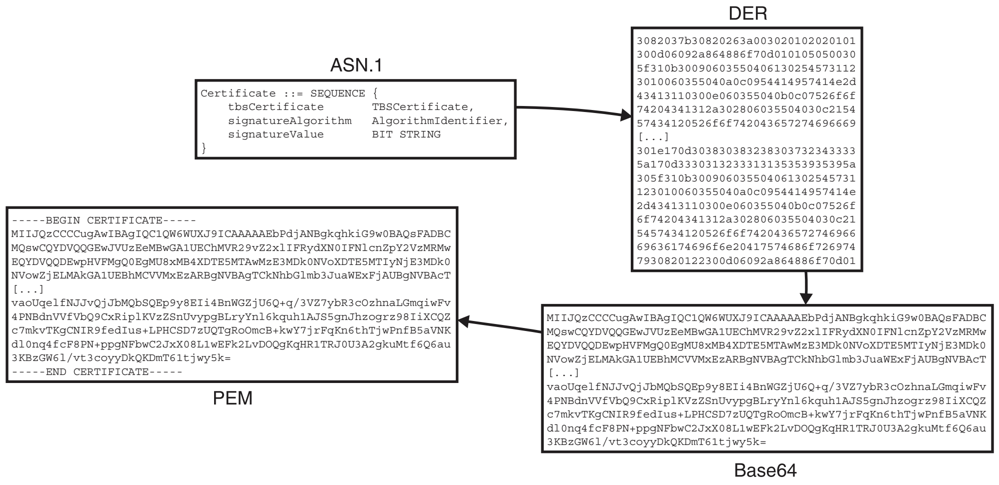
| | |
|---|---|
| **Subject Name** | |
| **Country or Region** | US |
| **State/Province** | California |
| **Locality** | Mountain View |
| **Organization** | Google LLC |
| **Common Name** | *.google.com |
| **Issuer Name** | |
| **Country or Region** | US |
| **Organization** | Google Trust Services |
| **Common Name** | GTS CA 1O1 |
| **Serial Number** | 0B 54 16 E9 65 17 27 D2 02 00 00 00 00 46 CF 76 |

OK

**Figure 9.7 Using Chrome's Certificate Viewer, we can observe the certificate chain sent by Google's server. The root CA is Global Sign, which is trusted by your browser. Down the chain, an intermediate CA called GTS CA 101 is trusted due to its certificate containing a signature from Global Sign. In turn, Google's leaf certificate, valid for *.google.com (google.com, mail.google.com, and so on), contains a signature from GTS CA 101.**

If you decode the base64 content surrounded by `BEGIN CERTIFICATE` and `END CERTIFICATE`, you end up with a *Distinguished Encoding Rules* (DER) encoded certificate. DER is a *deterministic* (only one way to encode) binary encoding used to translate X.509 certificates into bytes. All these encodings are often quite confusing to newcomers! I recap all of this in figure 9.8.

DER only encodes information as "here is an integer" or "this is a bytearray." Field names described in ASN.1 (like `tbsCertificate`) are lost after encoding. Decoding DER without the knowledge of the original ASN.1 description of what each field truly means is thus pointless. Handy command-line tools like OpenSSL allow you to decode and translate in human terms the content of a DER-encoded certificate. For example, if you download google.com's certificate, you can use the following code snippet to display its content in your terminal.



```
Certificate ::= SEQUENCE {
    tbsCertificate       TBSCertificate,
    signatureAlgorithm   AlgorithmIdentifier,
    signatureValue       BIT STRING
}
```

```
3082037b30820263a003020102020101
300d06092a864886f70d010105050030
5f310b30090603550406130254573112
3010060355040a0c0954414957414e2d
43413110300e060355040b0c07526f6f
74204341312a302806035504030c2154
57434120526f6f742043657274696669
[...]
301e170d303830383238303732343333
5a170d3330313233313135353935395a
305f310b300906035504061302545731
123010060355040a0c0954414957414e
2d43413110300e060355040b0c07526f
6f74204341312a302806035504030c21
5457434120526f6f7420436572746966
69636174696f6e20417574686f726974
7930820122300d06092a864886f70d01
```

```
MIIJQzCCCCugAwIBAgIQC1QW6WUXJ9ICAAAAAEbPdjANBgkqhkiG9w0BAQsFADBC
MQswCQYDVQQGEwJVUzEeMBwGA1UEChMVR29vZ2xlIFRydXN0IFNlcnZpY2VzMRMw
EQYDVQQDEwpHVFMgQ0EgMU8xMB4XDTE5MTAwMzE3MDk0NVoXDTE5MTIyNjE3MDk0
NVowZjELMAkGA1UEBhMCVVMxEzARBgNVBAgTCkNhbGlmb3JuaWExFjAUBgNVBAcT
[...]
vaoUqelfNJJvQjJbMQbSQEp9y8EIi4BnWGZjU6Q+q/3VZ7ybR3cOzhnaLGmqiwFv
4PNBdnVVfVbQ9CxRiplKVzZSnUvypgBLryYnl6kquh1AJS5gnJhzogrz98IiXCQZ
c7mkvTKgCNIR9fedIus+LPHCSD7zUQTgRoOmcB+kwY7jrFqKn6thTjwPnfB5aVNK
dl0nq4fcF8PN+ppgNFbwC2JxX08L1wEFk2LvDOQgKqHR1TRJ0U3A2gkuMtf6Q6au
3KBzGW6l/vt3coyyDkQKDmT61tjwy5k=
```

```
-----BEGIN CERTIFICATE-----
MIIJQzCCCCugAwIBAgIQC1QW6WUXJ9ICAAAAAEbPdjANBgkqhkiG9w0BAQsFADBC
MQswCQYDVQQGEwJVUzEeMBwGA1UEChMVR29vZ2xlIFRydXN0IFNlcnZpY2VzMRMw
EQYDVQQDEwpHVFMgQ0EgMU8xMB4XDTE5MTAwMzE3MDk0NVoXDTE5MTIyNjE3MDk0
NVowZjELMAkGA1UEBhMCVVMxEzARBgNVBAgTCkNhbGlmb3JuaWExFjAUBgNVBAcT
[...]
vaoUqelfNJJvQjJbMQbSQEp9y8EIi4BnWGZjU6Q+q/3VZ7ybR3cOzhnaLGmqiwFv
4PNBdnVVfVbQ9CxRiplKVzZSnUvypgBLryYnl6kquh1AJS5gnJhzogrz98IiXCQZ
c7mkvTKgCNIR9fedIus+LPHCSD7zUQTgRoOmcB+kwY7jrFqKn6thTjwPnfB5aVNK
dl0nq4fcF8PN+ppgNFbwC2JxX08L1wEFk2LvDOQgKqHR1TRJ0U3A2gkuMtf6Q6au
3KBzGW6l/vt3coyyDkQKDmT61tjwy5k=
-----END CERTIFICATE-----
```

**Figure 9.8 On the top left corner, an X.509 certificate is written using the ASN.1 notation. It is then transformed into bytes that can be signed via the DER encoding. As this is not text that can easily be copied around or recognized by humans, it is base64-encoded. The last touch wraps the base64 data with some handy contextual information using the PEM format.**

```
$ openssl x509 -in google.pem -text
Certificate:
    Data:
        Version: 3 (0x2)
        Serial Number:
            0b:54:16:e9:65:17:27:d2:02:00:00:00:00:46:cf:76
        Signature Algorithm: sha256WithRSAEncryption
        Issuer: C = US, O = Google Trust Services, CN = GTS CA 1O1
        Validity
            Not Before: Oct  3 17:09:45 2019 GMT
            Not After : Dec 26 17:09:45 2019 GMT
        Subject: C = US, ST = California, L = Mountain View, O = Google LLC,
CN = *.google.com
        Subject Public Key Info:
            Public Key Algorithm: id-ecPublicKey
                Public-Key: (256 bit)
                pub:
                    04:74:25:79:7d:6f:77:e4:7e:af:fb:1a:eb:4d:41:
                    b5:27:10:4a:9e:b8:a2:8c:83:ee:d2:0f:12:7f:d1:
                    77:a7:0f:79:fe:4b:cb:b7:ed:c6:94:4a:b2:6d:40:
                    5c:31:68:18:b6:df:ba:35:e7:f3:7e:af:39:2d:5b:
                    43:2d:48:0a:54
                ASN1 OID: prime256v1
                NIST CURVE: P-256
[...]
```

Having said all of this, X.509 certificates are quite controversial. Validating X.509 certificates has been comically dubbed "the most dangerous code in the world" by a team of researchers in 2012. This is because DER encoding is a difficult protocol to parse correctly, and the complexity of X.509 certificates makes for many mistakes to be potentially devastating. For this reason, I don't recommend any modern application to use X.509 certificates unless it has to.

#### Pre-shared keys and session resumption in TLS or how to avoid key exchanges

Key exchanges can be costly and are sometimes not needed. For example, you might have two machines that only connect to each other, and you might not want to have to deal with a public key infrastructure in order to secure their communications. TLS 1.3 offers a way to avoid this overhead with *pre-shared keys* (PSKs). A PSK is simply a secret that both the client and the server know, one that can be used to derive symmetric keys for the session.

In TLS 1.3, a PSK handshake works by having the client advertise in its ClientHello message that it supports a list of PSK identifiers. If the server recognizes one of the PSK IDs, it can say so in its response (the ServerHello message), and both can then avoid doing a key exchange (if they want to). By doing this, the authentication phase is skipped, making the Finished message at the end of the handshake important to prevent MITM attacks.

**Client random and server random**

An avid reader might have noticed that ephemeral public keys brought randomness to the session, and without them the symmetric session keys at the end of the handshake might end up always being the same. Using different symmetric keys for different sessions is extremely important as you do not want these sessions to be linked. Worse, because encrypted messages might be different between sessions, this could lead to nonce reuses and their catastrophic implications (see chapter 4).

To mitigate this, both the ClientHello and ServerHello messages have a `random` field, which is randomly generated for every new session (and often referred to as *client random* and *server random*). As these random values are used in the derivation of symmetric keys in TLS, it effectively randomizes the session symmetric keys for each new connection.

Another use case for PSKs is *session resumption*. Session resumption is about reusing secrets created from a previous session or connection. If you have already connected to google.com and have already verified their certificate chain, performed a key exchange, agreed on a shared secret, etc., why do this dance again a few minutes or hours later when you revisit? TLS 1.3 offers a way to generate a PSK after a handshake is successfully performed, which can be used in subsequent connections to avoid having to redo a full handshake.

If the server wants to offer this feature, it can send a New Session Ticket message at any time during the post-handshake phase. The server can create so-called *session tickets* in several ways. For example, the server can send an identifier, associated with the relevant information in a database. This is not the only way, but as this mechanism is quite complex and most of the time not necessary, I won't touch on more of it in this chapter. Next, let's see the easiest part of TLS—how communications eventually get encrypted.

### 9.2.2 How TLS 1.3 encrypts application data

Once a handshake takes place and symmetric keys derived, both the client and the server can send each other encrypted application data. This is not all: TLS also ensures that such messages cannot be replayed nor reordered! To do this, the nonce used by the authenticated encryption algorithm starts at a fixed value and is incremented for each new message. If a message is replayed or reordered, the nonce will be different from what is expected and decryption fails. When this happens, the connection is killed.

**Hiding the plaintext's length**

As you learned in chapter 4, encryption does not always hide the length of what is being encrypted. TLS 1.3 comes with *record padding* that you can configure to pad application data with a random number of zero bytes before encrypting it, effectively hiding the true length of the message. In spite of this, statistical attacks that remove the added noise may exist, and it is not straightforward to mitigate them. If you really require this security property, you should refer to the TLS 1.3 specification.

Starting with TLS 1.3, if a server decides to allow it, clients have the possibility to send encrypted data as part of their first series of messages, right after the ClientHello message. This means that browsers do not necessarily have to wait until the end of the handshake to start sending application data to the server. This mechanism is called *early data* or *0-RTT* (for zero round trip time). It can only be used with the combination of a PSK as it allows derivation of symmetric keys during the ClientHello message.

**NOTE** This feature was quite controversial during the development of the TLS 1.3 standard because a passive attacker can replay an observed ClientHello followed by the encrypted 0-RTT data. This is why 0-RTT must be used only with application data that can be replayed safely.

For the web, browsers treat every GET query as *idempotent*, meaning that GET queries should not change state on the server side and are only meant to retrieve data (unlike POST queries, for example). This is, of course, not always the case, and applications have been known to do whatever they want to do. For this reason, if you are confronted with the decision of using 0-RTT or not, it is simpler just to not use it.

## 9.3 The state of the encrypted web today

Today, standards are pushing for the deprecation of all versions of SSL and TLS that are not TLS versions 1.2 and TLS 1.3. Yet, due to legacy clients and servers, many libraries and applications continue to support older versions of the protocol (up to SSL version 3 sometimes!). This is not straightforward and, because of the number of vulnerabilities you need to defend against, many hard-to-implement mitigations must be maintained.

> **WARNING** Using TLS 1.3 (and TLS 1.2) is considered secure and best practice. Using any lower version means that you will need to consult experts and will have to figure out how to avoid known vulnerabilities.

By default, browsers still connect to web servers using HTTP, and websites still have to manually ask a CA to obtain a certificate. This means that with the current protocols, the web will never be fully encrypted, although some estimates show global web traffic to be 90% encrypted as of 2019.

The fact that, by default, your browser always uses an insecure connection is also an issue. Web servers nowadays usually redirect users accessing their pages using HTTP toward HTTPS. Web servers can also (and often do) tell browsers to use HTTPS for subsequent connections. This is done via an HTTPS response header called *HTTP Strict Transport Security* (HSTS). Yet, the first connection to a website is still unprotected (unless the user thinks about typing `https` in the address bar) and can be intercepted to remove the redirection to HTTPS.

In addition, other web protocols like *NTP* (to get the current time) and *DNS* (to obtain the IP behind a domain name) are currently largely unencrypted and vulnerable to MITM attacks. While there are research efforts to improve the status quo, these are attack vectors that one needs to be aware of.

There's another threat to TLS users—misbehaving CAs. What if, today, a CA decides to sign a certificate for your domain and a public key that it controls? If it can obtain a MITM position, it could start impersonating your website to your users. The obvious solution, if you control the client-side of the connection, is to either not use the web PKI (and rely on your own PKI) or to *pin* a specific certificate or public key.

Certificate or public key pinning are techniques where a server's certificate (usually rather a hash of it), or the public key, is directly hardcoded in the client code. If the server does not present the expected certificate, or the certificate does not contain the expected long-term public key, the client aborts the connection during the authentication phase of the handshake. This practice is often used in mobile applications, as they know exactly what the server's public key or certificate should look like (unlike browsers that have to connect to an infinite number of servers). Hardcoding certificates and public keys is not always possible, though, and two other mechanisms co-exist to deal with bad certificates:

- *Certificate revocation*—As the name indicates, this allows a CA to revoke a certificate and warn browsers about it.
- *Certificate monitoring*—This is a relatively new system that forces CAs to publicly log every certificate signed.

The story of certificate revocation has historically been bumpy. The first solution proposed was *Certificate Revocation Lists* (CRLs), which allowed CAs to maintain a list of revoked certificates, those that were no longer considered valid. The problem with CRLs is that they can grow quite large and one needs to constantly check them.

CRLs were deprecated in favor of *Online Certificate Status Protocol* (OCSP), which are simple web interfaces that you can query to see if a certificate is revoked or not. OCSP has its own share of problems: it requires CAs to have a highly available service that can answer to OCSP requests, it leaks web traffic information to the CAs, and browsers often decide to ignore OCSP requests that time out (to not disrupt the user's experience). The current solution is to augment OCSP with *OCSP stapling*: the website is in charge of querying the CA for a signed status of its certificate and attaches (staples) the response to its certificate during the TLS handshake. I review the three solutions in figure 9.9.

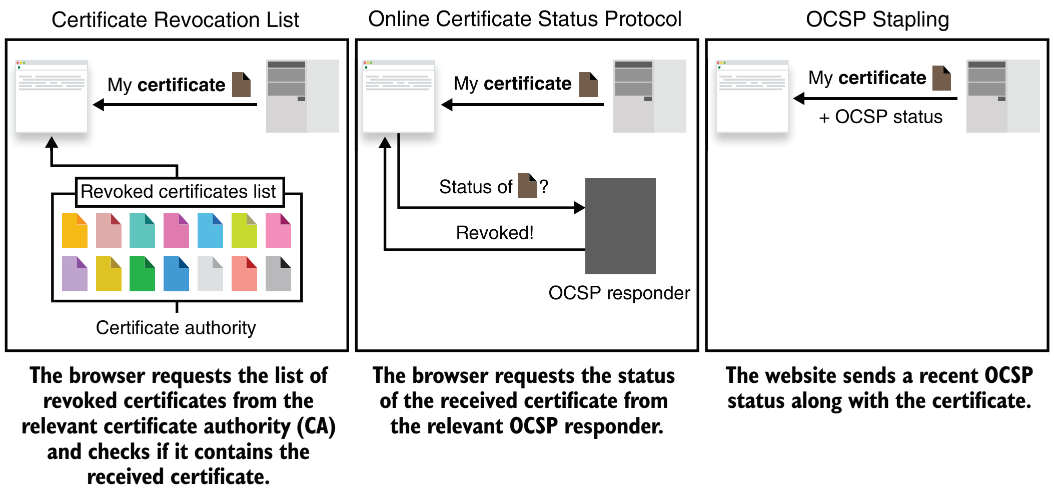

Certificate Revocation List
My certificate
Revoked certificates list
Certificate authority
The browser requests the list of revoked certificates from the relevant certificate authority (CA) and checks if it contains the received certificate.
Online Certificate Status Protocol
My certificate
Status of ?
Revoked!
OCSP responder
The browser requests the status of the received certificate from the relevant OCSP responder.
OCSP Stapling
My certificate
+ OCSP status
The website sends a recent OCSP status along with the certificate.

**Figure 9.9 Certificate revocation on the web has had three popular solutions: Certificate Revocation Lists (CRLs), Online Certificate Status Protocol (OCSP), and OCSP stapling.**

Certificate revocation might not seem to be a prime feature to support (especially for smaller systems compared to the World Wide Web) until a certificate gets compromised. Like a car seatbelt, certificate revocation is a security feature that is useless most of the time but can be a lifesaver in rare cases. This is what we in security call "defense in depth."

**NOTE** For the web, certificate revocation has largely proven to be a good decision. In 2014, the Heartbleed bug turned out to be one of the most devastating bugs in the history of SSL and TLS. The most widely used SSL/TLS implementation (OpenSSL) was found to have a *buffer overread* bug (reading past the limit of an array), allowing anyone to send a specially crafted message to any OpenSSL server and receive a dump of its memory, often revealing its long-term private keys.

Yet, if a CA truly misbehaves, it can decide not to revoke malicious certificates or not to report them. The problem is that we are blindly trusting a non-negligible number of actors (the CAs) to do the right thing. To solve this issue at scale, *Certificate Transparency* was proposed in 2012 by Google. The idea behind a Certificate Transparency is to force CAs to add each certificate issued to a giant log of certificates for everyone to see. To do this, browsers like Chrome now reject certificates if they do not include proofs of inclusion in a public log. This transparency allows you to check if a certificate was wrongly issued for a domain you own (there should be no other certificates other than the ones you requested in the past).

Note that a Certificate Transparency relies on people monitoring logs for their own domain to catch bad certificates *after the fact*. CAs also have to react fast and revoke mis-issued certificates once detected. In extreme cases, browsers sometimes remove misbehaving CAs from their trust stores. Certificate Transparency is, thus, not as powerful as certificate or public key pinning, which mitigates CA misbehaviors.

## 9.4 Other secure transport protocols

You've now learned about TLS, which is the most popular protocol to encrypt communications. You're not done yet, though. TLS is not the only one in the secure transport protocol class. Many other protocols exist, and you might most likely be using them already. Yet, most of them are TLS-like protocols, customized to support a specific use case. This is the case, for example, with the following:

- *Secure Shell (SSH)*—The most widely used protocol and application to securely connect to a remote terminal on a different machine.
- *Wi-Fi Protected Access (WPA)*—The most popular protocol to connect devices to private network access points or to the internet.
- *IPSec*—One of the most popular virtual network protocols (VPNs) used to connect different private networks together. It is mostly used by companies to link different office networks. As its name indicates, it acts at the IP layer and is often found in routers, firewalls, and other network appliances. Another popular VPN is OpenVPN, which makes direct use of TLS.

All of these protocols typically reimplement the handshake/post-handshake paradigm and sprinkle some of their own flavors on it. Re-inventing the wheel is not without issues, as for example, several of the Wi-Fi protocols have been broken. To finish this chapter, I want to introduce you to the *Noise protocol framework*. Noise is a much more modern alternative to TLS.

## 9.5 The Noise protocol framework: A modern alternative to TLS

TLS is now quite mature and considered a solid solution in most cases, due to the attention it gets. Yet, TLS adds a lot of overhead to applications that make use of it, due to historical reasons, backward compatibility constraints, and overall complexity.

Indeed, in many scenarios where you are in control of all the endpoints, you might not need all of the features that TLS has to offer. The next best solution is called the *Noise protocol framework*.

The Noise protocol framework removes the run-time complexity of TLS by avoiding all negotiation in the handshake. A client and a server running Noise follow a linear protocol that does not branch. Contrast this to TLS, which can take many different paths, depending on the information contained in the different handshake messages. What Noise does is that it pushes all the complexity to the design phase.

Developers who want to use the Noise protocol framework must decide what ad hoc instantiation of the framework they want their application to use. (This is why it is called a protocol *framework* and not a protocol.) As such, they must first decide what cryptographic algorithms will be used, what side of the connection is authenticated, if any pre-shared key is used, and so on. After that, the protocol is implemented and turns into a rigid series of messages, which can be a problem if one needs to update the protocol later while maintaining backward compatibility with devices that cannot be updated.

## 9.5.1 The many handshakes of Noise

The Noise protocol framework offers different *handshake patterns* that you can choose from. Handshake patterns typically come with a name that indicates what is going on. For example, the *IK* handshake pattern indicates that the client's public key is sent as part of the handshake (the first *I* stands for *immediate*), and that the server's public key is known to the client in advance (the *K* stands for *known*). Once a handshake pattern is chosen, applications making use of it will never attempt to perform any of the other possible handshake patterns. As opposed to TLS, this makes Noise a simple and linear protocol in practice.

In the rest of this section, I will use a handshake pattern called *NN* to explain how Noise works. It is simple enough for to explain, but insecure because of the two *N*'s indicating that no authentication takes place on both sides. In Noise's lingo, the pattern is written like this:

```
NN:
  -> e
  <- e, ee
```

Each line represents a message pattern, and the arrow indicates the direction of the message. Each message pattern is a succession of tokens (here, there are only two: `e` and `ee`) that dictates what both sides of the connection need to do:

- `-> e`—Means that the client must generate an ephemeral key pair and send the public key to the server. The server interprets this message differently: it must receive an ephemeral public key and store it.
- `<- e, ee`—Means that the server must generate an ephemeral key pair and send the public key to the client, then it must do a Diffie-Hellman (DH) key exchange

with the client's ephemeral (the first `e`) and its own ephemeral (the second `e`). On the other hand, the client must receive an ephemeral public key from the server, and use it to do a DH key exchange as well.

**NOTE** Noise uses a combination of defined tokens in order to specify different types of handshakes. For example, the `s` token means a *static key* (another word for *long-term key*) as opposed to an ephemeral key, and the token `es` means that both participants must perform a DH key exchange using the client's ephemeral key and the server's static key.

There's more to it: at the end of each message pattern (`-> e` and `<- e, ee`), the sender also gets to transmit a payload. If a DH key exchange has happened previously, which is not the case in the first message pattern, `-> e`, the payload is encrypted and authenticated. At the end of the handshake both participants derive a set of symmetric keys and start encrypting communications similarly to TLS.

### 9.5.2 A handshake with Noise

One particularity of Noise is that it continuously authenticates its handshake transcript. To achieve this, both sides maintain two variables: a hash (`h`) and a chaining key (`ck`). Each handshake message sent or received is hashed with the previous `h` value. I illustrate this in figure 9.10.

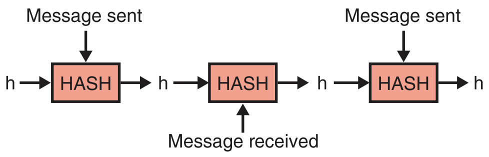


**Figure 9.10 In the Noise protocol framework, each side of the connection keeps track of a digest `h` of all messages that have been sent and received during the handshake. When a message is sent and encrypted with an authenticated encryption with associated data (AEAD) algorithm, the current `h` value is used as associated data in order to authenticate the handshake up to this point.**

At the end of each message pattern, a (potentially empty) payload is encrypted with an authenticated encryption with associated data (AEAD) algorithm (covered in chapter 4). When this happens, the `h` value is authenticated by the associated data field of the AEAD. This allows Noise to continuously verify that both sides of the connection are seeing the exact same series of messages and in the same order.

In addition, every time a DH key exchange happens (several can happen during a handshake), its output is fed along with the previous chaining key (`ck`) to HKDF, which derives a new chaining key and a new set of symmetric keys to use for authenticating and encrypting subsequent messages. I illustrate this in figure 9.11.

This makes Noise a simple protocol at run time; there is no branching and both sides of the connection simply do what they need to do. Libraries implementing Noise are also extremely simple and end up being a few hundred lines compared to hundreds of thousands of lines for TLS libraries. While Noise is more complex to use and

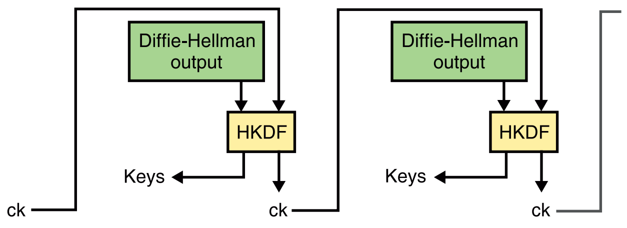


**Figure 9.11 In the Noise protocol framework, each side of the connection keeps track of a *chaining key*, `ck`. This value is used to derive a new chaining key and new encryption keys to be used in the protocol every time a DH key exchange is performed.**

will require developers who understand how Noise works to integrate it into an application, it is a strong alternative to TLS.

## Summary

- Transport Layer Security (TLS) is a secure transport protocol to encrypt communications between machines. It was previously called Secure Sockets Layer (SSL) and is sometimes still referred to as SSL.
- TLS works on top of TCP and is used daily to protect connections between browsers, web servers, mobile applications, and so on.
- To protect sessions on top of User Datagram Protocol (UDP), TLS has a variant called Datagram Transport Layer Security (DTLS) that works with UDP.
- TLS and most other transport security protocols have a handshake phase (in which the secure negotiation is created) and a post-handshake phase (in which communications are encrypted using keys derived from the first phase).
- To avoid delegating too much trust to the web public key infrastructure, applications making use of TLS can use certificate and public key pinning to only allow secure communications with specific certificates or public keys.
- As a defense-in-depth measure, systems can implement certificate revocation (to remove compromised certificates) and monitoring (to detect compromised certificates or CAs).
- In order to avoid TLS complexity and size and whether you control both sides of the connection, you can use the Noise protocol framework.
- To use Noise, one must decide what variant of a handshake they want to use when designing the protocol. Due to this, it is much simpler and secure than TLS, but less flexible.